Date 06/07/21

Meeraj Appari

A.I Practical-01

SDS Page No. Date

Aim - Write a program in Python to implement Breadth first search algorithm for romanian map and mumbai map problem

1) BFS description -
Breadth first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors and so on

* Strategies are evaluated along the following dimensions.

a) Completeness: Does it always find a solution if one exists

-> Yes (if b is finite), the shallowest solution is returned

b) Optimality - Does it always find atleast cost solution?

-> Yes, Solution quality is measured by the path cost function and a optimal solution has the lowest path cost among all solutions

c) Time complexity: Number of nodes generated

-> The total number of nodes generated is $b^2+b^3+\dots+b^d = O(b^d)$

d) Space complexity - Maximum number of nodes in memory

-> There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ node in the frontier. It keeps every node in memory

e) Time and space complexity are measured in terms of; the time is often measured in term

Neeraj Appa

SDS | Page No. | Date

number of node generated during the search and space in terms of maximum number of nodes stored in memory
f) b := maximum branching factor of the search tree
g) d := depth of the least-cost solution
h) m := maximum depth of the state space (may be ∞)
i) Search cost (time), total cost (time + space) search cost - which typical depends on the time complexity but can also include a term for memory usage or it can use the total cost which combines the search cost and the past cost of the solution found

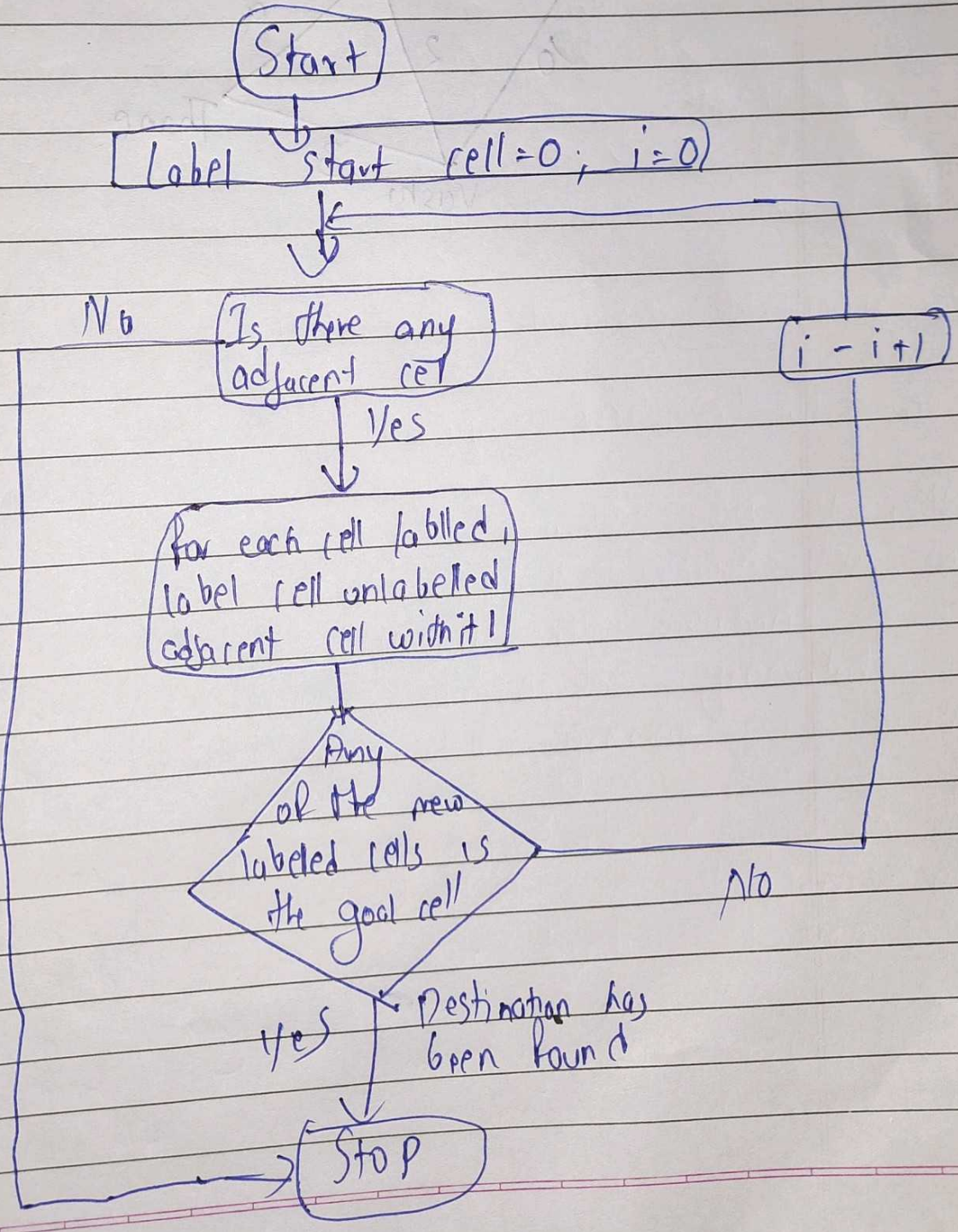
2] BFS Algorithm
1) import deque module
2) Make Class 'Node', self, state, parent, action, path cost
3) function 'repr' for to print node objects
4) function 'expand' to extract children
5) function 'childnod' to make node object of each child
6) function 'path' to extract the path of any node starting from current to source 'return list (reversed (path back)' order changed to show from source to current
7) function 'solution' to extract path of solution
8) Class graph problem it is the subclass of problem from function overridden
9) Class graph to represent graph

Neeraj Appan

SDS Page No.
Date / /

10) Breadth first search algorithm code
11) Print the value of mumbai map eg source to destination, path cost.
12) Romania Graph dictonary builded
13) Print the values of Romania map eg source to destination, path cost.

3) BFS Flowchart

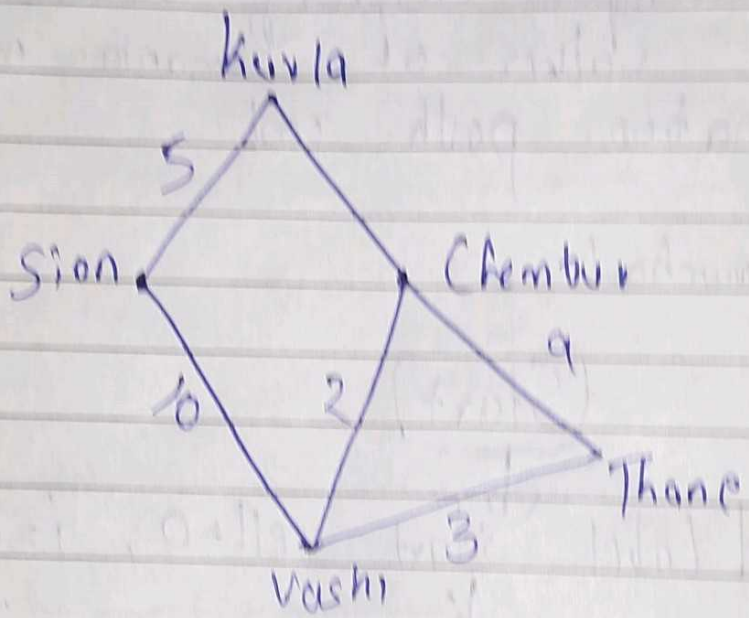


Aleena Appan TE

SDS | Page No.
Date / /

4) Draw the graph used in code

* Graph of Mumbai map



```
Neeraj_Appari
after construcing graph -
{'kurla': {'sion': 5, 'chembur': 6}, 'chembur':
    {'kurla': 6, 'thane': 9, 'vashi': 2},
    'vashi': {'sion': 10, 'chembur': 2, 'thane':
    3}, 'sion': {'kurla': 5, 'vashi': 10},
    'thane': {'chembur': 9, 'vashi': 3}}
------
['kurla', 'chembur', 'vashi', 'sion', 'thane']
Children of Kurla  {'sion': 5, 'chembur': 6}
distance from kurla to chembur=  6
============== BFS Algo ====================
Keys of kurla  ['chembur', 'vashi']
solution of  kurla  to  thane ['kurla',
    'chembur', 'thane']
path cost of final node = 15
solution of  Arad  to  Bucharest ['Arad',
    'Sibiu', 'Fagaras', 'Bucharest']
path cost of final node = 450
>
```

```
2  from collections import deque
3
4  infinity = float('inf')
5
6  class Node:
7      print("Neeraj_Appari")
8      def __init__(self, state, parent=None
           , action=None, path_cost=0):
9          self.state = state
10         self.parent = parent
11         self.action = action
12         self.path_cost = path_cost
13         self.depth = 0
14         if parent:
15             self.depth = parent.depth + 1
16
17     def __repr__(self): # to print node
           objects
18         return "<Node "+ self.state + ">"
19
20     def expand(self, problem): # to
           extract children
21         children = []
22         for action in problem.actions
               (self.state):
23             x=self.child_node(problem
                   ,action)
24             children.append(x)
25         return children
26
```

```
26
27
28      def child_node(self, problem, action
            ): # to make node object of each
            child
29          next_state = problem.result(self
                .state, action)
30          new_cost = problem.path_cost(self
                .path_cost, self.state,action
                , next_state)
31          #print("Current State = ", self
                .state, ",New Child = ",
                next_state,  ". Original path
                cost = ", self.path_cost, ",
                New path cost ", new_cost)
32          next_node = Node(next_state, self
                , action,new_cost )
33          return next_node
34
35      def solution(self): # extracts the
            path of solution
36          return [node.state for node in
                self.path()]
37
38      def path(self): # extracts the path
            of any node starting from current
            to source
39          node, path_back = self, []
40          while node:
41              path_back.append(node)
```

```
47      def __init__(self, initial, goal=None
            ):
48          self.initial = initial
49          self.goal = goal
50
51      def actions(self, state):
52          raise NotImplementedError
53
54      def result(self, state, action):
55          raise NotImplementedError
56
57      def goal_test(self, state):
58          return state == self.goal
59
60      def path_cost(self, c, state1, action
            , state2):
61          return c + 1
62
63      def value(self, state):
64          raise NotImplementedError
65
66
67  class GraphProblem(Problem): # subclass
        of problem, few functions overriden
68      def __init__(self, initial, goal,
            graph):
69          Problem.__init__(self, initial,
                goal)
70          self.graph = graph
```

```
65
66
67  class GraphProblem(Problem): # subclass
        of problem, few functions overriden
68      def __init__(self, initial, goal,
            graph):
69          Problem.__init__(self, initial,
                goal)
70          self.graph = graph
71
72      def actions(self, A):
73          return list(self.graph.get(A
                ).keys())
74
75      def result(self, state, action):
76          return action
77
78      def path_cost(self, cost_so_far, A,
            action, B):
79          return cost_so_far + (self.graph
                .get(A, B) or infinity)
80
81
82  class Graph: # to represent graph
83      def __init__(self, graph_dict=None,
            directed=True):
84          self.graph_dict = graph_dict or
                {}
85          self.directed = directed
86          if not directed:
```

```
80
81
82  class Graph: # to represent graph
83      def __init__(self, graph_dict=None,
            directed=True):
84          self.graph_dict = graph_dict or
                {}
85          self.directed = directed
86          if not directed:
87              self.make_undirected()
88
89      def get(self, a, b=None):
90          links = self.graph_dict.get(a)
91          if b is None:
92              return links
93          else:
94              cost = links.get(b)
95              return cost
96
97      def nodes(self):
98          nodelist = list()
99          for key in self.graph_dict.keys()
                :
100             nodelist.append(key)
101         return nodelist
102
103
104 def breadth_first_search(problem): #
        algorithm 3.11
105     node = Node(problem.initial)
```

```
106     if problem.goal_test(node.state):
107         return node
108     frontier = deque([node])
109     explored = set()
110     while frontier:
111         node = frontier.popleft()
112         explored.add(node.state)
113         for child in node.expand(problem
                ):
114             if child.state not in
                    explored and child not in
                    frontier:
115                 if problem.goal_test
                        (child.state):
116                     return child
117                 frontier.append(child)
118     return None
119
120
121 #we are giving full description of
      undireced graph through dictionary.
      the Graph class is not building any
      additional links
122
123 mumbaigraph=Graph({
124     'kurla':{'sion':5,'chembur':6},
125     'chembur':{'kurla':6,'thane':9,
          'vashi':2},
126     'vashi':{'sion':10,'chembur':
```

```
128     'thane':{'chembur':9,'vashi':3}
129     })
130
131 print("after construcing graph - ")
132 print(mumbaigraph.graph_dict)
133 print("------")
134 print(mumbaigraph.nodes() )
135 print("Children of Kurla ", mumbaigraph
        .get('kurla'))
136 print("distance from kurla to chembur= "
        ,mumbaigraph.get('kurla','chembur'))
137
138 print("=============== BFS Algo
        ====================")
139 mumbaigraph_problem = GraphProblem
        ('kurla','thane', mumbaigraph)
140 print("Keys of kurla ",
        mumbaigraph_problem.actions( 'thane'
        ))
141 finalnode = breadth_first_search
        (mumbaigraph_problem)
142 print("solution of ", mumbaigraph_problem
        .initial, " to ", mumbaigraph_problem
        .goal, finalnode.solution())
143 print("path cost of final node =",
        finalnode.path_cost)
144
145
146 romania_map = Graph(dict(
147     {'Arad': {'Zerind': 75. 'Sibiu': 140.
```

```
        'Fagaras': 211},
149     'Craiova': {'Drobeta': 120,
        'Rimnicu': 146, 'Pitesti': 138},
150     'Drobeta': {'Mehadia': 75, 'Craiova'
        : 120}, 'Eforie': {'Hirsova':
        86},
151     'Fagaras': {'Sibiu': 99, 'Bucharest'
        : 211},
152     'Hirsova': {'Urziceni': 98, 'Eforie'
        : 86},
153     'Iasi': {'Vaslui': 92, 'Neamt': 87},
154     'Lugoj': {'Timisoara': 111,
        'Mehadia': 70},
155     'Oradea': {'Zerind': 71, 'Sibiu':
        151},
156     'Pitesti': {'Rimnicu': 97,
        'Bucharest': 101, 'Craiova':
        138},
157     'Rimnicu': {'Sibiu': 80, 'Craiova':
        146, 'Pitesti': 97},
158     'Urziceni': {'Vaslui': 142,
        'Bucharest': 85, 'Hirsova': 98},
159     'Zerind': {'Arad': 75, 'Oradea': 71}
        ,
160     'Sibiu': {'Arad': 140, 'Fagaras': 99
        , 'Oradea': 151, 'Rimnicu': 80},
161     'Timisoara': {'Arad': 118, 'Lugoj':
        111},
162     'Giurgiu': {'Bucharest': 90}
        'Mehadia': {'Drobeta': 75
```

```
155         'Oradea': {'Zerind': 71, 'Sibiu':
                151},
156         'Pitesti': {'Rimnicu': 97,
                'Bucharest': 101, 'Craiova':
                138},
157         'Rimnicu': {'Sibiu': 80, 'Craiova':
                146, 'Pitesti': 97},
158         'Urziceni': {'Vaslui': 142,
                'Bucharest': 85, 'Hirsova': 98},
159         'Zerind': {'Arad': 75, 'Oradea': 71}
                ,
160         'Sibiu': {'Arad': 140, 'Fagaras': 99
                , 'Oradea': 151, 'Rimnicu': 80},
161         'Timisoara': {'Arad': 118, 'Lugoj':
                111},
162         'Giurgiu': {'Bucharest': 90},
                'Mehadia': {'Drobeta': 75,
                'Lugoj': 70},
163         'Vaslui': {'Iasi': 92, 'Urziceni':
                142}, 'Neamt': {'Iasi': 87}
164         }))
165 romania_problem = GraphProblem('Arad'
        ,'Bucharest', romania_map)
166 finalnode = breadth_first_search
        (romania_problem)
167 print("solution of ", romania_problem
        .initial, " to ", romania_problem
        .goal, finalnode.solution())
168 print("path cost of final node ="
        finalnode.path_cost)
```

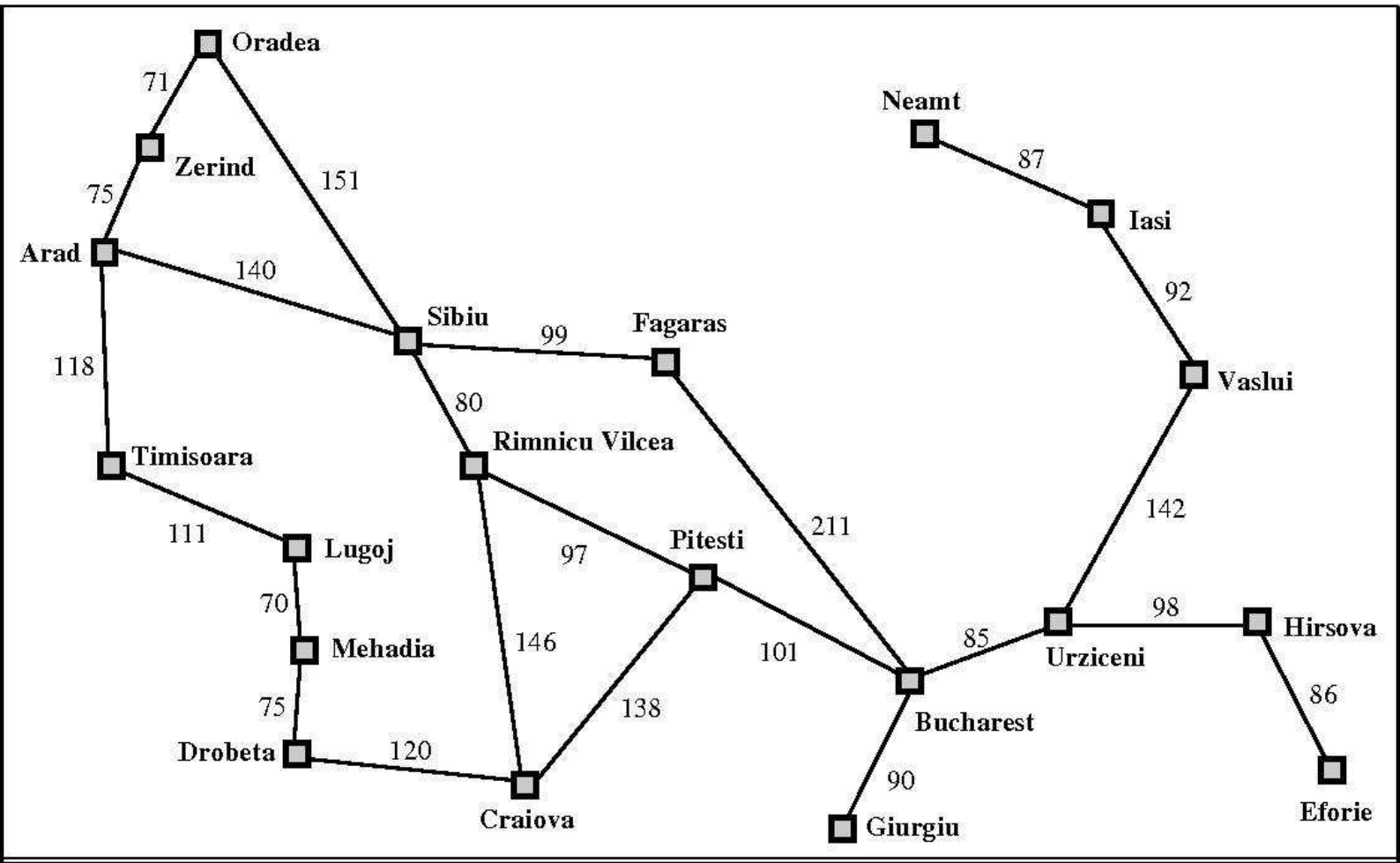


Figure 3.2 A simplified road map of part of Romania.